Dynamic Programming

Dynamic programming is a method used in mathematics and computer science to solve problems by breaking them down into simpler subproblems. It involves solving each subproblem only once and storing the solution in a data structure, typically a table, so that it can be easily retrieved later if needed.

This approach is often used when a problem can be divided into overlapping subproblems, and the solutions to those subproblems can be reused to solve the larger problem more efficiently.

## PROBLEM 01. Fibonacci number

### 

### Pseudocode (Naive Recursion):

|  |  |
| --- | --- |
| **Function** Fib(n):   1. **if** n==0 **or** n==1 **then** 2. **return** n 3. **end if** 4. **return** Fib(n-1) + Fib(n-2) | **Time complexity: O(2^n)**  **Space complexity: O(1)** |

### 

### DP Pseudocode (Memoization/Top-down):

|  |  |
| --- | --- |
| Create array F[0…n] = NIL  F[0]=0, F[1] = 1  **Function** Fib(n):   1. **if** n==0 **or** n==1 **then** 2. **return** n 3. **end if** 4. **if** F[n-1] = NIL **then** 5. F[n-1] = Fib(n-1) 6. **end if** 7. **if** F[n-2] = NIL **then** 8. F[n-2] = Fib(n-2) 9. **end if** 10. **return** F[n-1]+F[n-2] | **Time complexity: O(n)**  **Space complexity: O(n)** |

### 

### DP Pseudocode (Tabulation/Bottom-up):

|  |  |
| --- | --- |
| **Function** Fib(n):   1. Create array F[0…n] 2. F[0]=0, F[1] = 1 3. **for** i = 2 to n **do** 4. F[i] = F[i-1]+F[i-2] 5. **end for** 6. **return** F[n] | **Time complexity: O(n)**  **Space complexity: O(n)** |

## 

### DP Pseudocode (Problem-specific optimization):

|  |  |
| --- | --- |
| **Function** Fib(n):   1. f0=0, f1=1 2. **for** i = 2 to n **do** 3. f2 = f0+f1 4. f0 = f1 5. f1 = f2 6. **end for** 7. **return** f1 | **Time complexity: O(n)**  **Space complexity: O(1)** |

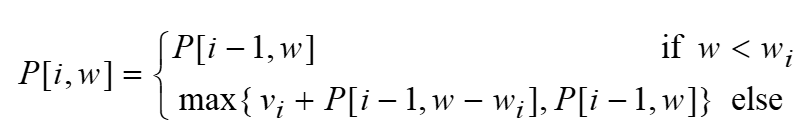
## 

## PROBLEM 02. 0-1 Knapsack

The weights and values of **n** items are given. *The items are not divisible, i.e., you cannot take a fraction of an item.*You have a knapsack to carry those items, whose weight capacity is **W**. Due to the capacity limit of the knapsack, it might not be possible to carry all the items at once. In that case, pick items such that the profit (total values of the taken items) is maximized.

Write a program that takes the weights and values of **n** items, and the capacity **W** of the knapsack from the user and then finds the items which would maximize the profit using a dynamic programming algorithm.

|  |  |
| --- | --- |
| **sample input**  n  weight, value  …  W | **sample output** |
| 4  4 20  3 9  2 12  1 7  5 | item 1: 4.0 kg 20.0 taka  item 4: 1.0 kg 7.0 taka  profit: 27 taka |



### Pseudocode (Tabulation method):

|  |  |
| --- | --- |
| **Function** Knapsack( v[], w[], W): | **Time complexity: O(nW)**  **Space complexity: O(nW)** |

### Food for thought:

### How to print the items taken?

### Pseudocode (More detail version):

Function knapsack(weights[], values[], n, capacity):

dp[n+1][capacity+1] //2D array to store the maximum value

For i from 0 to n:

For j from 0 to capacity:

dp[i][j] = 0 // Initialize the dp array with zeros

For i from 1 to n: // Fill the dp array

For j from 1 to capacity:

If weights[i-1] > j:

dp[i][j] = dp[i-1][j] // If the current item cannot be included

Else:

dp[i][j] = max(dp[i-1][j], values[i-1] + dp[i-1][j - weights[i-1]])

// Traverse the dp array to find the items included in the knapsack

Let selected\_items = []

i = n

j = capacity

While i > 0 and j > 0:

If dp[i][j] != dp[i-1][j]:

selected\_items.append(i) // Include the current item

j = j - weights[i-1] // Move to the previous item

i = i - 1

Return dp[n][capacity], selected\_items

## PROBLEM 03. Longest common subsequence

Given two strings x and y, find the longest common subsequence and its length.

*Example:*

x = “A**BCB**D**A**B”

y = “**B**D**C**A**BA**”

longest common subsequence = “BCBA”

longest common subsequence length = 4

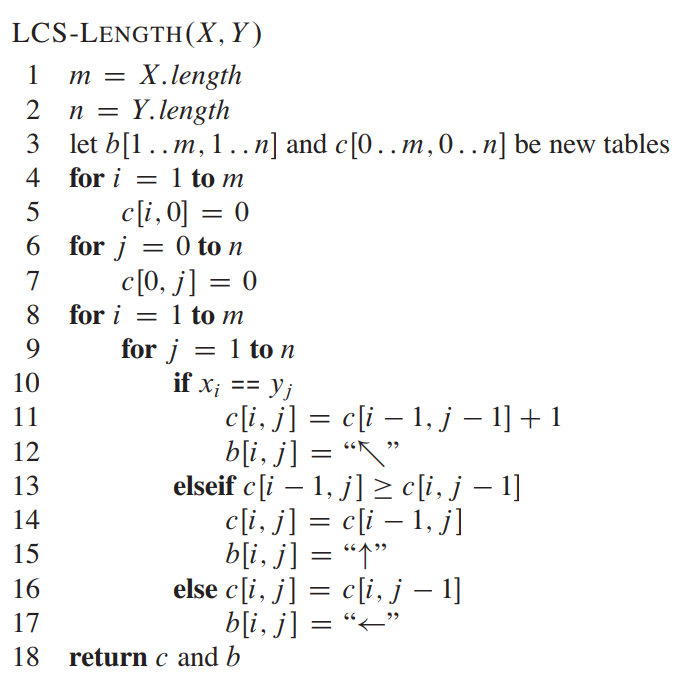
x = “**AB**B**ACQ**”

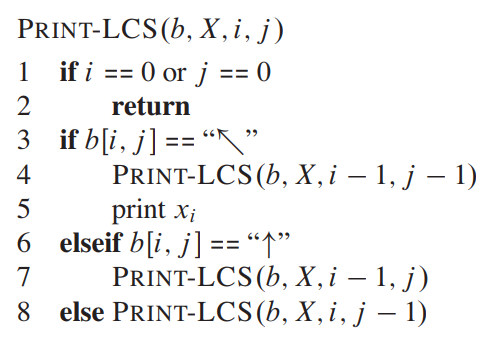
y = “X**A**YZM**B**NN**A**LQ**C**TR**Q**”

longest common subsequence = “ABACQ”

longest common subsequence length = 5

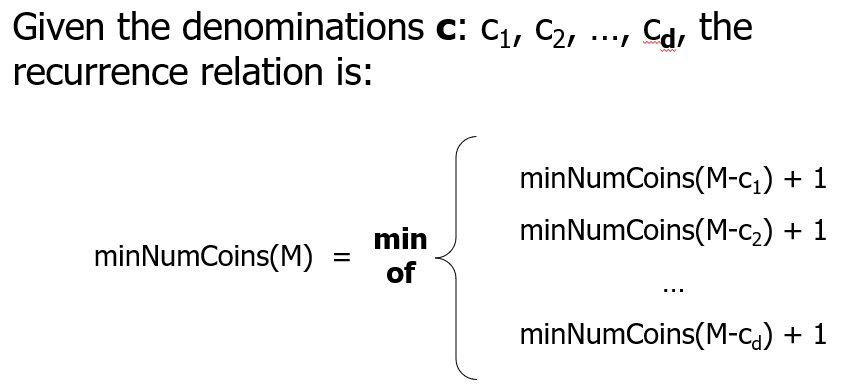
Hint: CLRS 15.4





## PROBLEM 04. Coin change problem

Consider the problem of making change for **M** cents using the fewest number of coins. There are **d** types of coins **C = {c1, c2, …, cd}**, each coin’s value is an integer and there are an infinite number of coins for each coin type. Write a greedy algorithm to make change consisting of coins in **C**.



### Pseudocode (Naive Recursion):

|  |
| --- |
| **Function** Recursive-Change (*M, C, d*):   1. **if** M = 0 **then** 2. **return** 0 3. **end if** 4. mnc = infinity // minimum number of coins 5. **for** i = 1 to d **do** 6. **if** C[i] <= M **then** 7. nc = Recursive-Change(M-C[i], C, d) 8. **if** nc+1 < mnc **then** 9. mnc = nc+1 10. **end if** 11. **end if** 12. **end for** 13. **return** mnc |

### 

### Pseudocode (Tabulation method):

|  |
| --- |
| **Function** Coin-Change (*M, C, d*):   1. create an array mnc[0…M] 2. mnc[0] = 0 3. **for** m = 1 to M **do** 4. mnc[m] = infinity 5. **for** i=1 to d **do** 6. **if** C[i] <= m **and** mnc[m-C[i]]+1 < mnc[m] **then** 7. mnc[m] = mnc[m-C[i]]+1 8. **end if** 9. **end for** 10. **end for** 11. **return** mnc[M] |

### Pseudocode (Broad Version)

function minCoins(coins[], amount):

n = coins.length

// Create a 2D array to store results of subproblems

dp[][] = new Array(n+1, amount+1)

// Initialize the base cases

for i from 0 to n:

dp[i][0] = 0

for j from 0 to amount:

dp[0][j] = infinity

// Fill the dp table

for i from 1 to n:

for j from 1 to amount:

// If the current coin denomination is greater than the current amount,

// we cannot include it, so copy the previous row's value

if coins[i-1] > j:

dp[i][j] = dp[i-1][j]

else:

// Otherwise, take the minimum of (not using current coin, 1 + using current coin)

dp[i][j] = min(dp[i-1][j], 1 + dp[i][j - coins[i-1]])

return dp[n][amount]

### 

### Food for thought:

How to print the coins taken?

## Alternative Problem 2.2

Consider the problem of making a **M** meter long rope using smaller ropes. There are **d** types of ropes **C = {c1, c2, …, cd}**, each rope’s value is an integer and there are an infinite number of ropes for each rope type. Joining two ropes together costs X dollar. Write an algorithm to make the **M** meter long rope with minimum costs.